MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Spring Semester, 1996

Problem Set 8

Issued: April 4, 1996

Written solutions due: Wednesday, April 17 for EVEN numbered sections; Friday, April 19 for ODD numbered sections.

Reading: Through Section 3.3.3, plus attached code (`world.scm` and `game.scm`).

Note: The overall object-oriented programming framework will be discussed in lecture on Thursday, April 4.

The programming assignment for this week explores two ideas: the simulation of a world in which objects are characterized by collections of state variables, and the use of *object-oriented programming* as a technique for modularizing worlds in which objects interact. These ideas are presented in the context of a simulation, much as might be used by economists, meteorologists, political scientists or physicists to analyze the complex interactions implied by mathematical models of the real world. Our system is a cross between these "serious" simulations and the popular simulation games (e.g, textual adventure games) available on many computers. While playing these games can be quite fun, programming them can be difficult if little or no effort is put into understanding the system's design and implementation. It is therefore very important that you adequately study the system and plan your work before coming to the lab.

Part 1 will help you to master the ideas involved. Part 2 consists of some tutorial exercises that you should complete before visiting the lab (you may wish to double check them using the machines in the lab). Part 3 contains a few warm-up exercises to do in the lab. You might do these and check your understanding of the answers before proceeding to the main lab assignment itself in Part 4.

Part 1: The 6.001 Election Game

The basic idea of a simulation is that the user creates a scenario in an imaginary world inhabited by various objects with special properties that govern how they interact. The user runs the simulation by creating appropriate (simulated) objects located in appropriate (simulated) locations with differing sets of parameters selected to test a model's ability to predict behavior under different conditions. In addition, some objects in the simulation may be under direct user control as the simulation proceeds. The user issues commands to the computer that have the effect of moving the objects or changing parameters in the imaginary world, such as picking up objects. The computer simulates the legal moves and rejects illegal ones. For example, it is illegal to move between places that are not connected (unless you have special powers). If a move is legal, the computer updates its model of the world and allows the next move to be considered.

Our game takes place in the strange world of U.S. politics. This world is inhabited by politicians, voters, reporters, and special investigators. In order to get going, we need to establish the structure of this imaginary world: the objects that exist and the ways in which they relate to each other.

Initially, there are three procedures for creating objects:

```
(make-city name)
(make-voter city how-influencable talkative?)
(make-politician name city risk-aversion restlessness)
```

In addition, there are procedures that make people and things, and procedures that install them in the simulated world. The reason that we need to be able to create people and things separately from installing them will be discussed in one of the exercises later. For now, we note the existence of the procedures

```
(make&install-city name)
(make&install-voter city how-influencable talkative?)
(make&install-politician name city risk-aversion restlessness)
```

All objects in the system are implemented as message-accepting procedures, as described in lecture on April 4. In addition to the main objects of the simulation, the file `game.scm` defines a number of other types of objects which are needed to construct cities, voters, and politicians. This is analogous to the "class library" that comes with most object-oriented development environments.

Our simulation has two kinds of people: voters and politicians. They have much in common (in fact, `person` is exactly what they have in common). We've built our world so that all people are located somewhere (in a `place`), but only politicians have names and can move about from place to place. The behavior of a politician is controlled by two variables (in our imaginary world, anyway): how much `risk-aversion` they have (0 means they don't like to take risks, 1 means they take a risk whenever they can) and their `restlessness` (a non-negative integer).

Voters are anonymous (they don't have names), but they do have a location. In addition, voters have two properties: they can be influenced to a certain extent (rated between 0 – not influencable at all and 1 – very easily persuaded) and a boolean that controls how much output they generate as the game proceeds.

The file `world.scm` has code to create a simulated world. It begins by creating objects to represent the major cities of the United States (Cambridge, Palo Alto, El Paso, Fairbanks, etc.) It then connects the cities to simulate airplane routes (we used the same techniques that the airlines apparently use: we selected locations at random and connected them with non-stop flights). Your major job will be to create objects that simulate the airplanes; but we'll get to that later.

We then create a very complicated type of person, the `*the-registrar-of-voters*`. This is by far the most complicated object in our system, but you don't need to worry about how it's built. Just notice the messages you can send to it: `REGISTER-CANDIDATE`, `REGISTER-VOTER`, `ELECTION`, plus the usual messages that any `person` will accept[1].

Finally, we create the voters and the politicians. The number of voters in each city is chosen somewhat randomly (between 10 and a selectable maximum number). As the voters are created,

[1] Actually, `*the-registrar-of-voters*` accepts three other messages: `tally`, `merge-results`, and `report-results` but these aren't part of the object's external interface.

they are registered with the `*the-registrar-of-voters*`. The politicians are scattered around the cities, with randomly selected restlessness and risk-aversion factors.

After loading `game.scm` followed by `world.scm` the simulation is ready to go.

Part 2: Tutorial exercises

Tutorial exercise 1: (a) Define a procedure `flip` (with no parameters) that returns 1 the first time it is called, 0 the second time it is called, 1 the third time, 0 the fourth time, and so on. (b) Now, define a procedure `make-flip` that can be used to generate procedures that have the same behavior as `flip` from part (a). That is, if we had `make-flip` for part (a), we could have implemented `flip` as `(define flip (make-flip))`. (c) Draw an environment diagram to illustrate the result of evaluating the following sequence of expressions:

```
(define (make-flip) ...)
(define flip1 (make-flip))
(define flip2 (make-flip))

(flip1) --> value?
(flip2) --> value?
```

Tutorial exercise 2: Assume that the following definitions are evaluated, using the procedure `make-flip` from the previous exercise:

```
(define flip (make-flip))
(define flap1 (flip))
(define (flap2) (flip))
(define flap3 flip)
(define (flap4) flip)
```

What is the value of each of the following expressions (evaluated in the order shown)?

```
flap1

flap2

flap3

flap4

(flap1)

(flap2)

(flap3)

(flap4)

flap1

(flap3)

(flap2)
```

Tutorial exercise 3: Consider the following code very carefully:

```
(define (get-method message preferred . others)
  ;; Get the "best" method, assuming objects are ordered from best to
  ;; worst.  GET-METHOD-FROM-OBJECT is in file game.scm.
  (define (loop objs)
    (let ((method (get-method-from-object message (car objs)))
          (rest (cdr objs)))
      (if (or (method? method) (null? rest))
          method
          (loop rest))))
  (loop (cons preferred others)))
```

Write a very short paragraph (no more than four sentences) describing how this code works. You might look at the notes or past problem sets for examples of good descriptive style; it should be correct, concise, and complete. Your description should, in particular, explain why it returns `method` when `(null? rest)` is true, even if `method` isn't really a method.

Tutorial exercise 4: The lab is much easier to do if you have a complete list of the types of objects and the class structure (which object inherits from which other). Create a sheet with this information on it, bring it with you, and make a copy for your tutor. There are two ways to prepare this information, and the choice is yours. The first is to draw a table with rows for data types and columns for messages, with a check mark if an object of the data type can handle the message. You can alternatively draw a directed graph with data types for nodes, arrows for showing which type inherits from which other, and with each data-type indicating the messages it can handle by itself. In the latter case, the root of the tree would be `named-object` (with messages `NAME`, `INSTALL`, and `SAY`).

Tutorial exercise 5: Suppose we evaluate the following expressions:

```
(define taxes (make-mobile-object 'student-money Cambridge))
(ask taxes 'CHANGE-LOCATION WashingtonDC)
```

At some point in the evaluation of the second expression, the expression

```
(set! location new-place)
```

will be evaluated in some environment. Draw an environment diagram, showing the full structure of `taxes` at the point where this expression is evaluated. Don't show the details of `Cambridge` or `WashingtonDC`—just assume that `Cambridge` and `WashingtonDC` are names defined in the global environment that point off to some objects that you draw as blobs.

Tutorial exercise 6: Suppose that, in addition to `taxes` in exercise 5, we define

```
(define local-taxes (make-mobile-object 'student-money cambridge))
```

Are `taxes` and `local-taxes` the same object (i.e., are they `eq?`)? What will result if we install `taxes` and `local-taxes` into the same location?

Part 3: Lab Warm-up Exercise

Lab Warm-up 1: Ask the politician `test-pol` and the city `Cambridge` for their names (both are defined in `world.scm`). Write a procedure that takes an object that inherits from (delegates to) `physical-object` and returns the name of that object's location. Test it by showing the name of the city in which the `test-pol` is located. Turn in the code and a transcript demonstrating that it works.

Lab Warm-up 2: Before trying out the simulation, let's take a look at it a bit. Notice that a politician, if it can't understand a message, delegates it to a `traveller`; and a `traveller` has a message `TELEPORT` which makes it choose a city at random and move there. So, we should be able to move our politicians around by sending them a `TELEPORT` message. In our current implementation, however, politicians are very close-mouthed about their movements. Let's fix that.

Change the code for `make-politician` so that it will print a single message when it receives a `TELEPORT` message. The message should inform us of the politician's location before and after teleportation. Turn in a listing of the `method` that implements the change, along with a short transcript showing that it works. Here's some possible output[2]:

[2] If you want to make the exact changes shown here, you will have to think very carefully about how `politicians`, `travellers`, and `persons` interact. In particular, think about how to get the politician to act like a `person` when (s)he speaks but as a `mobile-object` otherwise. Of course, it's hardly a surprise that it's hard to make a politician act like a person ...

```
(ask test-pol 'teleport)
At fairbanks : test says -- Teleported from fairbanks to cambridge
;Value: nuf-said
(ask test-pol 'teleport)
At cambridge : test says -- Teleported from cambridge to washingtondc
;Value: nuf-said
```

In principle, you could run the system by issuing specific commands to each of the objects in the world, but this defeats the intent of the game since that would give you explicit control over all the objects[3]. Instead, we will structure our system so that any object can be manipulated automatically in some fashion by the computer. We do this by creating a list of all the objects to be moved by the computer and by simulating the passage of time by a special procedure, `clock`, that sends a `clock-tick` message to each object in the list.

Different objects in our simulation react differently to the passage of time. In our simulation up to this point, only politicians and cities react to a clock tick.

Lab Warm-up 3: Louis Reasoner claims that having a separate `INSTALL` method for objects is unnecessary. He claims that the `INSTALL` method should be called automatically every time an object is created. For example, Louis proposes that `make-politician` be modified as follows:

```
(define (make-politician
           name initial-location thrill-seeking restlessness)
  (let ((traveller (make-traveller name initial-location))
        (ticks-to-go restlessness))
    (define politician                ;Louis added this line
      (lambda (message)
        (case message
          ...
          (else (get-method message traveller)))))
    (ask politician 'INSTALL)         ;Louis added this line
    politician))                      ;Louis added this line
```

Other make procedures would be modified the same way. Alyssa P. Hacker, however, disagrees. She says that Louis's implementation would leave "ghost" objects behind when politicians left cities.

Who is right? Provide a brief explanation of your answer.

Lab Warm-up 4: Write a very brief description (less than 3 sentences) describing what a politician does on each clock tick; be sure to explain what the `restlessness` argument to `make-politician` does. Do the same for a city.

Now it's time to run the simulation at full speed for a while. The procedure `run-clock` will run the simulation for a specified number of clock ticks.

[3]Besides, who types faster: you or the computer?

Lab Warm-up 5: Try running the simulation for, say, 5 ticks. Then hold an election by evaluating the expression `(ask *the-registrar-of-voters* 'ELECTION)`. For how many ticks did you have to run the simulation in order to have a clear winner after only one round of voting? Turn in a transcript of your interaction with the system[4].

Part 4: Lab exercises

When you load the code for problem set 8, the system will load `game.scm`. We do not expect you to have to make significant changes in this code, though you may do so if you want to.

The system will also set up a buffer with `world.scm` and load it into SCHEME. Since the simulation model works by data mutation, it is possible to get your SCHEME-simulated world into an inconsistent state while debugging. To help you avoid this problem, we suggest the following discipline: any procedures you change or define should be placed in your answer file; any new characters or objects you make and install should be added to `world.scm`. This way whenever you change some procedure you can make sure your world reflects these changes by simply re-evaluating the entire `world.scm` file. Finally, to save you from retyping the same scenarios repeatedly—for example, when debugging you may want to create a new character, move it to some interesting place, then ask it to act—we suggest you define little test "script" procedures at the end of `world.scm` which you can invoke to act out the scenarios when testing your code. See the comments in `world.scm` for details.

Lab exercise 1: Meta-adventure You can inspect an environment structure using the `show` procedure from `game.scm`. `Show` is a bit like the `pp` procedure you have used in the past for printing out procedures, but it prints things out so that they look more like parts of an environment diagram. It can be used like this:

```
(show cambridge)
#[compound-procedure 40]
Frame:
  #[environment 41]
Body:
  (lambda (message)
    (cond ((eq? message 'clock-tick) (lambda ... ...))
          ((eq? message 'install) (lambda ... ... ...))
          ((eq? message 'sponsor-debate) (lambda ... ...))
          ...))
```

Now you can inspect the environment of the procedure by calling `show` with the 'hash number' of the environment (41, in this example). The hash number is the number after '`compound-procedure`' or '`environment`' in the usual printed representations of these objects. The system guarantees that all different (i.e. non-`eq?`) objects have different hash numbers so you can tell if you get back to the same place.

[4]This is intended to be easy – just run the simulation and see what happens. Don't try to analyze your result or find a repeatable answer to this question!

```
(show 41)
#[environment 41]
Parent frame: #[environment 42]
place:        #[compound-procedure 43]
```

Here we see that the environment frame that is part of the procedure `cambridge` has one variable defined in it (`place`) and, of course, has a parent frame (number 42).

This exercise is called meta-adventure because you are going to use the `show` procedure to explore and 'map' the environment structure for `cambridge` and produce an environment diagram.

Start with `cambridge` and follow all the hash numbers *except* those of other cities, politicians, or voters (just show their names). There should be about 10 things to show. Print out the results and cut out the individual results. Arrange the pieces on a large blank piece of paper so that they are in the correct positions to make an environment diagram. Glue the pieces in place and draw in the arrows to make a complete environment diagram. Turn in your diagram.

Warning: The environment in this game can get very large in a hurry. Thus we strongly suggest that you start with a clean Scheme system, and just load `game.scm` and `world.scm`, then answer this question.

Making an airplane: the framework

Right now, our politicians move about using `teleport`. This is a delightful way to travel, but not currently available. It is also quite risky, since there is no way to know where you'll go when you teleport (the technology isn't quite fully developed yet). In the real world, most politicians travel between large cities by airplane. So lets add the simulation of airplanes to our model world.

We'll start by thinking about the state variables that describe an airplane. Then we'll see how we can (re-)use existing objects to implement the behavior of an airplane, and only after we've exhausted that will we write our own code for the plane. Let's assume that each plane has a unique flight number (that will help debugging, even if it's a bit unrealistic), and a schedule that includes a certain amount of time on the ground followed by a certain amount of flying time. In addition, it needs a clock to indicate where it is in the cycle of `(+ flying-time ground-time)` clock ticks that it needs to make a trip. That gives us a reasonable starting template (you will find this in the file `plan.scm`):

```
(define make-plane
  (let ((flight 0))
    (lambda (from to flying-time)
      (define (random-number n)
        ;; Generate a random number between 1 and n
        (+ 1 (random n)))
      (let ((ground-time (random-number 4))
            (my-flight-no flight)
            (current-time 0))
        (define (plane message)
          (case message
            ((INSTALL)
             (lambda (self) ...))
            ((AIRPLANE?) (lambda (s) true))
            ((DESTINATION) (lambda (s) to))
            ((FLIGHT) (lambda (s) my-flight-no))
            ((CLOCK-TICK)
             (lambda (self) ...))
            (else (get-method message ...))))
        (set! flight (+ flight 1))
        plane))))
(define (make&install-plane from to flying-time)
  (make&install-object make-plane from to flying-time))
```

Lab exercise 2: There's no good reason for treating ground time as a random number rather than an input parameter to `make-plane`; it was just a design choice. But this design choice is reflected in code somewhere else in the system. If we decided to make it an input parameter, what other piece(s) of code would have to change?

Lab exercise 3: Why do we have both the variable `flight` and `my-flight-no`? If we removed `my-flight-no` from the `let` and replaced all other occurrences with `flight` what would happen?

Lab exercise 4: Write the code for `INSTALL`. It should pick a random number between 1 and `(+ flying-time ground-time)` for the current time (notice that you have a utility procedure named `random-number` to help with this). If this number is greater than the `ground-time`, then the airplane is flying, so its location should be `*the-sky*`. Don't worry about the fact that airplanes don't have locations yet; we'll fix that in a moment. Just assume that they can accept the appropriate message. Also, add the new plane to the clock list so that it will receive `CLOCK-TICK` messages. Turn in your implementation of the `INSTALL` method.

Lab exercise 5: Write and turn in the code for `CLOCK-TICK`. This is an exercise in wishful thinking. Basically, the plane should work as follows:

- At times $0 < t <$ `ground-time` the plane waits on the ground. We simulate this with a message (sent by the plane to itself) called `WAIT-ON-GROUND`.
- At time $t =$ `ground-time`, the plane departs. We can simulate this with the message `DEPART`.
- At times `ground-time` $< t <$ `ground-time`+`flying-time` the plane is flying; use the message `FLY`.

- When $t =$ `ground-time+flying-time`, the plane lands; use the message `ARRIVE`.

After sending the appropriate message to itself, the plane should update its internal clock by adding one (but don't forget to wrap around from `ground-time+flying-time` to 1).

Inheriting Behavior by Delegating Messages

With the basic code for creating an airplane in place, it is time to decide just how our airplane fits into the object framework. If you think about it carefully you will realize that an airplane is a `mobile-object` (since it moves from city to city) as well as a `place` (since people[5] can be located on the airplane).

Lab exercise 6 Modify the airplane code you created above so that it can handle all of the messages sent to either a `mobile-object` or to a `place`. Is a plane more like one than the other (i.e. does it matter which is the "preferred" type of object in the call to `get-method`)? Don't forget to modify the `INSTALL` handler (method) you wrote so that it installs any new objects that it creates.

Handling Internal Messages

To complete the airplane simulation, we need code to handle the four new messages that are sent by the airplane to itself when the clock ticks (`ARRIVE`, `DEPART`, `WAIT-ON-GROUND`, and `FLY`).

Lab exercise 7 The basic work of these four handlers is very simple. There is no required work to do when the airplane receives the messages `WAIT-ON-GROUND` or `FLY`. When the airplane receives the message `DEPART` it must change its own location to be `*the-sky*`, and the message `ARRIVE` must make the plane change its location to the destination city. For reasons we'll see below, the `ARRIVE` message should also send a `DEPLANE` message to each passenger (that means you'll need to implement a `DEPLANE` message for politicians before you test your code out!).

But you should use your imagination to define the handlers for these four messages. You might want them to print out messages that will help you understand what the simulation is doing. Later on, you may want to add new ideas (hijacking, emergency landing, delayed departure, etc.)

Turn in your completed code for `make-plane`.

Lab exercise 8 The implementation of airplanes is now complete. You should test it by writing a script which will create an airplane, place a politician on it, and then just let the clock tick until the plane takes off, lands, and deposits the politician in the new location. Turn in your script and a transcript that shows it working. Run a few tests of your own, too, just to make sure that everything is working OK.

[5] Well, politicians anyway.

"Improving" the Politician

Now that we have invented airplanes, it's time to have our politicians use them. To start, replace the handler for the politician's `TRAVEL` message with the following:

```
((TRAVEL)
 (lambda (self)
   (if (weighted-choice thrill-seeking)
       (ask self 'TELEPORT)
       (let ((city
              (pick-random
               (ask (ask self 'LOCATION)
                    'NEIGHBORS))))
         (if city (ask self 'FLY city))))))
```

The idea is that a thrill-seeking politician will always travel by teleporting, and a stodgy politician will only take airplanes. When travelling by plane, we have to pick a city and the politician does that at random (you might want to put in a better strategy later: like keeping track of each city's population and going to the neighboring city with the highest ratio of residents to politicians currently visiting there).

This is all well and good, but how should politicans handle the `FLY` message? The basic answer is "they shouldn't!" That's what a `traveller` is for: it understands (and implements) all of the modes of transportation. All we need to do is implement a new method, `FLY`, in the traveller and the problem is solved. Sort of. We also have to be careful to update the way it handles the message `TRAVELLING` – previously, a traveller could simply respond `#F` to this message, since teleportation is instantaneous. But airplanes take time, so we have to model that, as well.

The basic idea will be to implement `FLY` as follows:

1. Travellers are travelling if they are either waiting for a plane to arrive at the city they are currently in, or they are on a plane and waiting for it to land.
2. As long as they are waiting for a plane, travellers will check to see what planes are currently at their city. If there are any that are going to their destination they will hop aboard and will continue to wait until they land at the destination.
3. When a plane lands, it will send a `DEPLANE` message to the passengers. You implemented a version of `DEPLANE` for politicians before (in lab exercise 7); you may have to remove it in order to finish this exercise.

Lab exercise 9 In order to implement the `FLY` message it will be useful to have a procedure, `All-Planes-To`, which takes two arguments: a `mobile-object` and a `place`. It returns a list of all of the planes that are at the same location as the `mobile-object` and are destined for the specified `place`. Turn in a listing of your procedure. Show how it works by looking for all of the airplanes to Cambridge from wherever `test-pol` is currently. Move `test-pol` to several different cities to make sure that your procedure is working correctly.

Lab exercise 10 Implement the `FLY` message for your `traveller`. You may do this in any way you like, but one way to do it involves:

- adding new state variables `flying?`, `waiting-for-plane?`, and `flying-to`. The handler for each message is responsible for updating these variables as they change.
- adding a new message, `FIND-PLANE-TO`, which receives a destination as an argument. It uses `All-Planes-To` to find all of the planes at the current location with the desired destination, and chooses one at random to board.
- having the handler for the `FLY` message send itself a `FIND-PLANE-TO` message.
- having the handler for `TRAVELLING?` message call `FIND-PLANE-TO` if the traveller is trying to catch a plane.
- having the handler for `DEPLANE` change the location of the traveller to the final destination (passed as an argument to the message).

Optional Part 5: Contest

This part of the lab is completely optional. You may make any set of modifications you like to the lab. Turn in both the modified code and a demonstration of how the code works in practice.

Prizes will be awarded for the cleverest ideas turned in for this problem. You might consider such things as implementing `reporter` objects (maybe they try to get interviews instead of staging debates, or maybe they influence voters) and/or investigators (do they try to convince voters to vote against someone?). Have fun. It's a presidential election year!